Electronic Communications of the EASST
Volume 47 (2012)

Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2012)

Layout Improvement in Diagram Editors by Automatic Ad-hoc Layout

Sonja Maier and Mark Minas

14 pages

Guest Editors: Andrew Fish, Leen Lambers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Layout Improvement in Diagram Editors by Automatic Ad-hoc Layout

Sonja Maier[1] **and Mark Minas**[2]

[1] sonja.maier@unibw.de [2] mark.minas@unibw.de
Universität der Bundeswehr München, Germany

Abstract: Layout, in the context of diagram editors, is the positioning of diagram components on the screen. Editor users enjoy automatic layout, but they usually like to control the layout at runtime, too. Our pattern-based layout approach allows for automatic and user-controlled layout at the same time: The diagram editor may automatically apply layout patterns to diagram parts based on syntactic rules provided by the editor developer, but editor users may also select diagram parts and then apply layout patterns to them. For instance, user-selected components may be aligned horizontally and remain aligned even after diagram modifications.

This paper describes continued work on pattern-based layout. We present automatic ad-hoc layout which combines automatic and user-controlled layout in a new way. While automatic layout is syntax-based and must be specified by the editor developer in advance, automatic ad-hoc layout is solely based on the current diagram layout. Whenever the layout engine detects a situation where a pattern may be applied with no or only small diagram changes, this layout pattern is automatically applied. For instance, if a set of components is almost horizontally aligned on the screen, the horizontal alignment pattern is automatically applied to these components. Such an editor behavior is known from so-called snap lines in commercial diagram editors. Automatic ad-hoc layout generalizes on these manually programmed layout solutions and offers many additional layout features.

This paper describes the concept of automatic ad-hoc layout as well as its integration into a diagram editor framework and discusses issues of this new layout approach.

Keywords: layout, pattern, meta-model

1 Introduction

A layout engine usually runs continuously within diagram editors and improves the layout in response to user interaction in real-time. Layout improvement includes all sorts of changes concerning the position or shape of diagram components. For instance, in Figure 1, if node `F` is moved, the end point of the connected edge is updated accordingly.

On one side, it is not reasonable that the layout engine completely automatically arranges a diagram. On the other side, it does not make sense that the editor user completely has to arrange a diagram himself. Instead, the editor user wants some sort of automatic layout improvement, which he can influence at runtime.

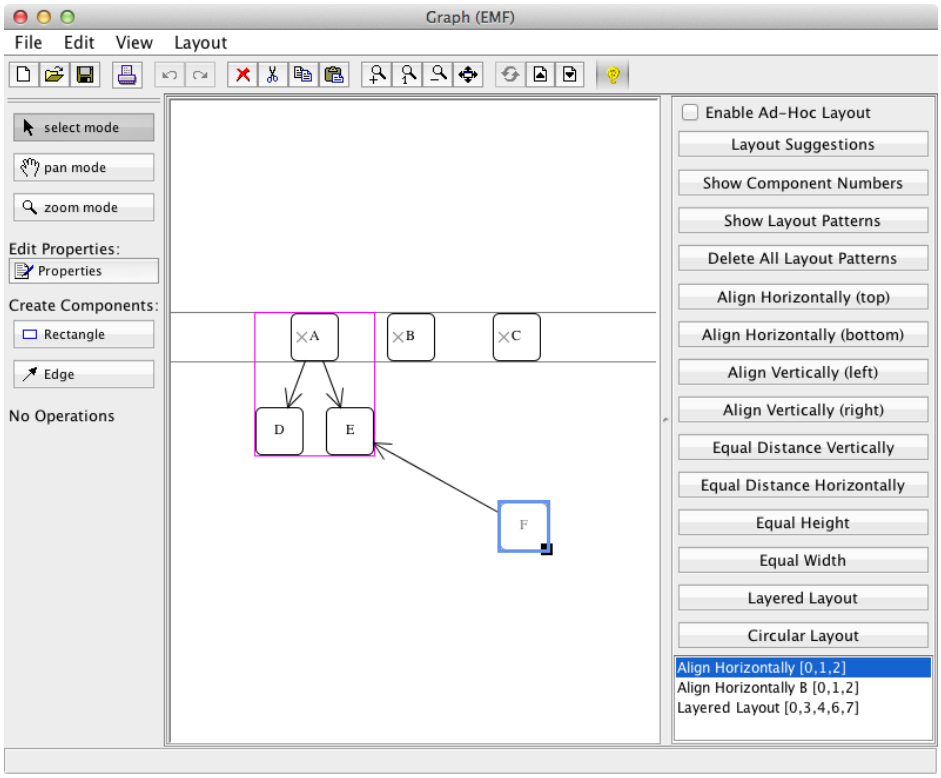


Figure 1: Simple Graph Editor

We have developed a *pattern-based layout approach* [MM10b] that is tailored to interactive environments. Layout patterns allow to encapsulate many different kinds of layout algorithms and makes them easily reusable for many different types of diagrams. The approach depends on a *control algorithm* [MM10a] for computing a diagram layout that is determined by different layout pattern instances. An overview of the whole approach outlined below and its integration into editors is provided in [MM12].

The layout of a diagram is defined by applying layout patterns to selected sub-diagrams. Applying a pattern to a sub-diagram means creating a *pattern instance* and binding it to this sub-diagram. The pattern instance then contributes to automatic layout, i.e., its layout algorithm adjusts component attributes according to the pattern's specification. So far, we have distinguished two modes of operation which can be used simultaneously in the same diagram: *Automatic application* selects layout patterns and sub-diagrams automatically, controlled by the specification of the diagram syntax. For *user-controlled application*, the editor user selects diagram components that he would like to be arranged according to a layout pattern and applies the corresponding pattern to the sub-diagram consisting of this set of components.

This paper introduces *automatic ad-hoc layout* as a third mode of operation. We first recap *layout suggestions* that have been briefly introduced in [MM12]. When the editor selects some diagram components, the layout engine automatically suggests what layout patterns may be applied to the selected sub-diagram. *Automatic ad-hoc layout* even goes one step further: The

layout engine autonomously looks for extensions of the current set of selected components and automatically applies layout patterns if appropriate. For instance, if selected components are almost horizontally aligned with other components, the horizontal alignment pattern is automatically applied to these components, i.e., they are aligned automatically. Such an editor behavior is known from so-called snap lines in commercial diagram editors.

The rest of the paper is structured as follows: Section 2 discusses some related work. Section 3 introduces a graph editor, which is used as a running example, and describes the layout functionality associated with this editor. Our layout approach is sketched in Section 4. Section 5 describes automatic ad-hoc layout. This mode of operation makes new functionality possible which is discussed in Section 6. Section 7 concludes the paper.

2 Related Work

There exist some diagram editors that are somewhat related to our approach. The most relevant examples are GLIDE [RMS97] and DUNNART [DMW09, DMS+08], two editors for graph-based visual languages. Both editors provide some user-controlled layout behavior, and their layout computation is based on declarative constraints. Our approach provides a comparable user-controlled layout behavior. The editors mentioned also have in common with our approach that layout computation is based on some sort of constraints, which are called assertions in the following. One of the main differences is the underlying constraint solver that computes a valid layout. Another difference is that our approach provides an additional level of abstraction: Associated assertions are bundled as so-called layout patterns. With the concept of layout patterns, the specification of layout behavior on an abstract level is straightforward, and the reuse of layout behavior in different editors is enabled.

Many commercial tools, such as POWERPOINT, VISIO or OMNIGRAFFLE allow for some sort of user-controlled layout behavior. E.g., components can be aligned horizontally. However, alignment is not permanently enforced, only when the user selects this layout action. We distinguish permanently enforced layout from such layout actions applied only once. Our approach supports both variants, whereas commercial tools mainly support single layout actions and support permanently enforced layout in a rather restricted way.

There exist many layout engines that provide the possibility of automatic graph layout. Two prominent examples are ZEST[1], which is included in GEF, and YFILES[2], which is a well know graph drawing library. These layout engines are best suited for graph-based visual languages, whereas our approach supports all kinds of visual languages. They are usually used for the visualization of diagrams, and allow the user to influence the layout in a very restricted way.

The feature *snap to grid* [BS86] is included in many tools: After each user interaction, the diagram components are moved such that they are "snapped" to a grid. A generalized version of this feature is hypersnapping [Mas01]: After each user interaction, the diagram components are moved such that they are "snapped" to a certain object.

In the paper "Grids and Guides: Multi-Touch Alignment Tools" [FKLD11], Frisch et al. present some features that belong to the category of automatic layout improvement, which can

[1] eclipse.org/gef/zest
[2] www.yworks.com

be controlled by the user. Their approach uses some sort of hypersnapping where components are bound to shapes, e.g., to lines or circles. In their work, they mainly focus on the use of such features on multi-touch screens.

3 Running Example: Graph Editor

Figure 1 shows a graph editor,[3] which has been created with the editor generation framework DIAMETA [Min06]. DIAMETA allows for generating visual language editors from specifications of their visual languages. The core of the specification is a meta-model called *language-specific meta-model* (LMM). This meta-model is created with EMF [SBPM09] and comprises two parts, the *abstract syntax meta-model*, representing the language's abstract syntax, and the *concrete syntax meta-model*, representing the language's concrete syntax.

Several layout patterns have been integrated into the editor. A brief description of each of these patterns can be found in Table 1. As previously mentioned, instances of some patterns are created automatically, considering the whole diagram. In our example, these are the patterns non-overlap, minimal size, and edge follower. Instances of the other patterns are created by the user, considering a user-selected part of the diagram. In our example, these are the patterns alignment, equal distance, equal size, and layered layout.

Table 1: Layout Patterns in the Graph Editor

Pattern	**Description**
Non-Overlap	Removes overlapping of nodes by performing force-directed layout.
Minimal Size	A minimal size of nodes is enforced.
Edge Follower	Makes sure that edges stay attached to nodes.
Alignment (H & V)	Aligns certain nodes vertically or horizontally respectively.
Equal Distance (H & V)	Makes sure that certain nodes have an equal distance to each other.
Equal Size (W & H)	Makes sure that nodes have the same height or width respectively.
Layered Layout	Assigns each node to a horizontal layer. The "correct" layer is determined by an examination of the nodes and edges present in the diagram. E.g., the Sugiyama algorithm [BETT99] may be used.

4 Pattern-based Layout Approach

Our layout approach is based on the concept of layout patterns, which was introduced in [MM10b]. Each layout pattern consists of a *pattern-specific meta-model (PMM)*, some *assertions* on attribute values of components specified by the PMM, and a *layout algorithm* that modifies component positions if assertions are violated. That way, layout patterns can encapsulate a variety of different kinds of layout algorithms. So far, we have used the following types of layout algorithms:

[3] Screencasts that show the editor are available at www.unibw.de/inf2/DiaGen/Layout.

- *Standard graph drawing algorithms* such as the Sugiyama algorithm [BETT99].
- *Constraint-based algorithms* which compute the layout by the help of a numeric constraint solver. With this type of layout algorithm, global layout behavior such as "several nodes have an equal distance to each other" can be easily specified.
- *Rule-based layout specification* uses a simple constraint solver which is specifically tailored to the interactive nature of visual language editors [MM07]. As a precursor of our layout patterns, such a specification consists of a set of assertions that "define" the layout, and a set of transformation rules that "repair" violated assertions and the layout after user modifications.

A layout pattern p is applied to a sub-diagram consisting of a set C of diagram components by creating an instance $\mathscr{I}(p,C)$ of the pattern. This means that the sub-diagram's model is transformed into a *pattern-specific model (PM)* that conforms to the PMM. Of course, a pattern can only be applied to sub-diagrams whose model can be transformed to a PM. The pattern's assertions constrain attributes within the PM and, therefore, of the sub-diagram the pattern is applied to. The set of all pattern instances of a diagram thus distinguishes a valid from an invalid layout of the diagram. It is the task of the patterns' layout algorithms to transform an invalid layout to a valid one. This is done by executing the layout algorithm of those pattern instances whose assertions are violated. This means that all layout algorithms of those pattern instances must be arranged in a sequence and then executed along this sequence. Finding a sequence such that all assertions are satisfied in the end is not trivial. In [MM10a], we have presented a control algorithm that finds such a sequence using a backtracking approach.

As outlined in Section 1, our pattern-based layout approach enables *automatic* and *user-controlled application* of patterns.

Automatic instantiation means that patterns are automatically applied depending on the diagram syntax. For that purpose, patterns are specified together with the diagram language specification, i.e., the meta-model for abstract and concrete syntax. When the editor analyzes the syntactic structure of the diagram, the corresponding patterns are automatically applied to the corresponding sub-diagrams.

An alternative to automatic application is **user-controlled application**. By selecting some diagram components and a layout pattern, the user chooses a sub-diagram for which the pattern is applied. After creation, the pattern instance continuously contributes to the whole diagram's layout until the editor user explicitly deletes this pattern instance again.

In the example shown in Figure 1, the user has created three pattern instances: The nodes `A`, `D`, and `E` together with the two corresponding edges are rearranged by the Sugiyama algorithm (called *layered layout algorithm* in Figure 1). The nodes `A`, `B`, and `C` are aligned horizontally at the top *and* at the bottom. In addition, some pattern instances were automatically created that also update the layout of the diagram. When the user modifies the diagram, the alignment pattern instance and the layered layout pattern instance preserve the layout that was chosen by the user. In addition, the non-overlap pattern instances move the nodes to assure that they do not overlap. The minimal size pattern instances enforce a minimal size of the nodes. The edge follower pattern instances update the start and end point of the edges to keep them correctly connected to the nodes.

Case studies have shown that editor users have troubles remembering the currently active pattern instances and "understanding" the layout dependencies between components after applying layout patterns if pattern instances are not displayed in the diagram. In order to avoid such a confusion, diagram editors can display currently active layout pattern instances. In the example, instances of the horizontal alignment pattern are displayed via gray lines, and instances of the layered layout pattern via colored boxes. Different instances of the same layout pattern are distinguished by different colors.

If the user selects one of the pattern instances in the list at the bottom-right of the editor, the corresponding components are highlighted via a gray cross in the middle of each component. In the editor shown in Figure 1, the user has selected one of the alignment pattern instances for the nodes `A`, `B`, and `C` which are highlighted.

5 Automatic Ad-hoc Layout

The concept of layout patterns allows for further increasing the usability of the editors. In the following, we describe layout suggestions that the layout engine can compute automatically. Based on these layout suggestions, we then describe automatic ad-hoc layout which allows for the automatic application of layout patterns solely based on the current diagram and without specification with the diagram syntax.

5.1 Layout Suggestions

The pattern-based layout approach allows for further user support: When the user selects a set of diagram components and, therefore, defines a sub-diagram, the layout engine can suggest those patterns from the set of all available patterns that may be applied to the sub-diagram. Computing these layout suggestions has been sketched in [MM12] already and is straight-forward: The set of all layout suggestions initially consists of the set of all available layout patterns. First, each pattern is removed from the set of suggestions that cannot be applied to the selected sub-diagram because the pattern either does not fit the chosen sub-diagram or is inconsistent with the currently active pattern instances. Next, each of the remaining patterns is "tried" to be applied to the selected sub-diagram, i.e., the layout engine computes the layout modifications that would be necessary if the patterns were applied. Only those patterns are kept in the set of suggestions that would require a small layout modification, the others are removed from the set. We use a rather simple metrics for deciding whether a layout modification is small as described in the following. Note that layout suggestions do not change the diagram layout but just suggest those patterns that are appropriate for the selected sub-diagram.

For measuring the size of a layout modification, we compute the mean square of all attribute changes as follows: Let C_s be the set of components selected by the user and $A(c)$ the set of all attributes of a component $c \in C_s$. Furthermore, for any component $c \in C_s$ and any attribute $a \in A(c)$, let $\mathrm{val}_{\mathrm{prev}}(a)$ and $\mathrm{val}(a)$ be the attribute value of a before and after computing the layout modification. We then compute the size d of the layout modification by

$$d = \frac{\sum_{c \in C_s} \sum_{a \in A(c)} (\mathrm{val}(a) - \mathrm{val}_{\mathrm{prev}}(a))^2}{\sum_{c \in C_s} |A(c)|}.$$

A layout modification is considered *small* if $d < m$ for some threshold m.

In a diagram editor, layout suggestions can be used as follows: The layout engine computes the layout suggestions when the user has selected diagram components and then pushes the button *Layout Suggestions*. The layout suggestions are displayed by highlighting the buttons on the right side of the editor in a certain color. Black indicates that the corresponding pattern cannot be applied to the selected components because it either does not fit the chosen diagram part or is inconsistent with the currently active pattern instances. The other buttons are drawn in blue. Stars are added to the button labels of the patterns within the set of layout suggestions, i.e., whose application would result in a small layout modification.

Figure 2 shows the results when computing the layout suggestions for the nodes `X`, `Y`, and `Z`. All buttons are drawn in blue since each pattern can be applied. But not all of them would result in small layout modifications, e.g., aligning these three nodes vertically. Align Vertically (left), therefore, is not a layout suggestion and not marked by stars, whereas Align Horizontally (top) is a layout suggestion since the three nodes are almost aligned horizontally already. This button, therefore, is marked with stars.

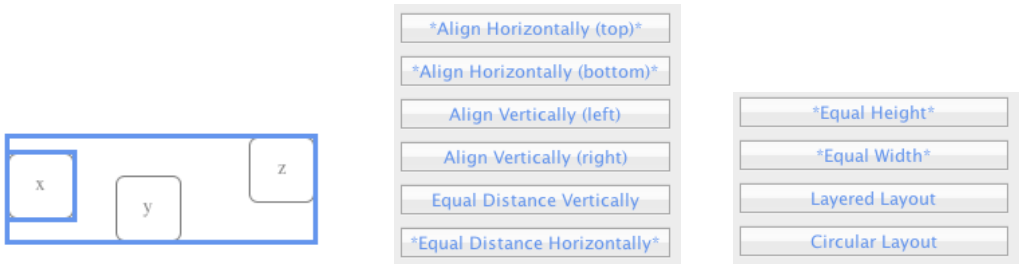


Figure 2: Layout suggestions indicated by the diagram editor

So far, layout suggestions have been computed based on some user-selected diagram parts. Automatic ad-hoc layout as a more powerful mode of operation is made possible by allowing the layout engine to autonomously extend the set of selected components. The following sections restrict these extensions in different ways, enabling different kinds of ad-hoc layout.

5.2 Global Ad-hoc Layout

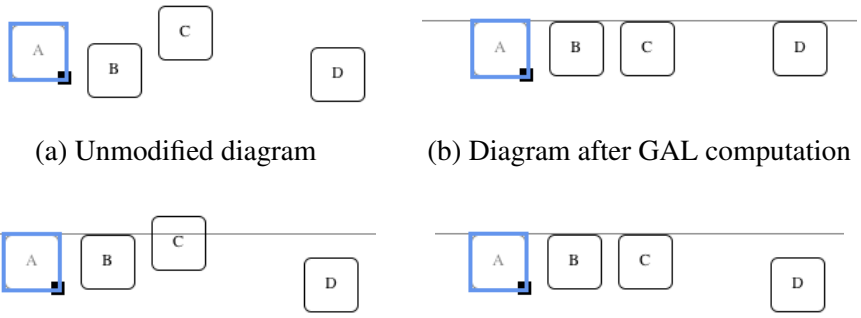
Global Ad-hoc Layout (GAL) means that layout suggestions are computed and automatically applied for all elementary extensions C of the set C_s of user-selected components, i.e., for each set $C \subseteq C_{all}$ such that $|C \setminus C_s| = 1$ where C_{all} indicates the set of all diagram components.

The algorithm is outlined in Listing 1. It gets as input the set P of all layout patterns available in the diagram editor, the set C_s of selected components, and the set C_{all} of all components (formal parameter C_{max}). It returns *true* iff it has applied a pattern. The algorithm may influence the diagram layout globally because it considers every diagram component.

E.g., in the example shown in Figure 3(a), the user has moved the component A. As a consequence, layout suggestions are computed and applied for the sets $\{A,B\}$, $\{A,C\}$, and $\{A,D\}$ of components. The horizontal alignment pattern is instantiated for these sets because the attribute changes after application are quite small as can be seen in Figure 3(b).

```
proc computeAdHocLayout(P, C_s, C_max)
  candidates := P × (C_max \ C_s)
  do
    instances := ∅
    for each (p, c) ∈ candidates do
      if p applied to C_s ∪ {c} results in small layout modifications then
        apply p to C_s ∪ {c}
        add (p, c) to instances
      end if
    end do
    candidates := candidates \ instances
  while instances ≠ ∅
  return candidates ≠ P × (C_max \ C_s)
end
```

Listing 1: Algorithm for computing GAL



(a) Unmodified diagram (b) Diagram after GAL computation

(c) Diagram after LAL computation (d) Diagram after LAL/P computation

Figure 3: Automatic Ad-hoc Layout

5.3 Local Ad-hoc Layout

GAL must try layout patterns for all diagram components, even those far apart from the user-selected ones. Therefore, performance is an issue, and it may lead to some surprising layout modifications at distant diagram locations. *Local ad-hoc layout (LAL)* enhances on these aspects. In contrast to GAL, LAL computes layout suggestions only for components that are in a close neighborhood of the selected components. These components form the set $C_n \subseteq C_{all}$ of components. More precisely, only components $c \in C_{all}$ that are "near" the set C_s of selected components are considered. For that purpose, the minimal distance between the bounding box of the component c and the bounding box of each of the selected components $c_s \in C_s$ is computed. A component is considered being near the selected components if this distance is less than a certain threshold t:

$$C_n = \{c \in C_{all} \mid \exists c_s \in C_s : \mathrm{dist}(c, c_s) < t\}$$

LAL can again use the algorithm in Listing 1. However, it is called with the neighborhood C_n instead of the set C_{all} of all components.

E.g., in the example shown in Figure 3(a), the user has moved the component A. The neighborhood of C_s is just $C_n = \{A, B\}$ since the components C and D are too far apart. LAL then suggests and applies the horizontal alignment pattern for the set $\{A, B\}$ of components. The result is shown in Figure 3(c).

5.4 Local Ad-hoc Layout with Propagation

As the example shows, LAL has only small benefits. *Local ad-hoc layout with propagation (LAL/P)* enhances on this aspect. It starts with the set $C = C_s$ of user-selected components and computes the local ad-hoc layout for C. The set C is then extended by all components that have just been modified, and local ad-hoc layout is computed again for this extended set C of components. This iteration is continued until no further pattern instance has been created. This way, layout improvement is "propagated" through the diagram, as long as new layout suggestions can be computed. The propagation algorithm is outlined in Listing 2.

E.g., in the example shown in Figure 3(a), the user has moved the component A. As described before, local ad-hoc layout suggests and applies a layout modification horizontally aligning A and B whereas components C and D are too far apart. Since component B has just been changed, local ad-hoc layout is applied again for the set $C = \{A, B\}$ of components with a neighborhood $C_n = \{A, B, C\}$. This time, C is horizontally aligned to A and B by local ad-hoc layout. The next iteration with the set $C = \{A, B, C\}$ has the neighborhood $C_n = \{A, B, C\}$ again since D is located too far apart. No suggestions are computed, and the algorithm stops. The result is shown in Figure 3(d).

The example above shows that the layout engine cannot just add new pattern instances to the set of active pattern instances: The first iteration added a pattern instance $i_1 = \mathscr{I}(p_A, \{A, B\})$, and the second iteration $i_2 = \mathscr{I}(p_A, \{A, B, C\})$ where p_A indicates the alignment pattern. Pattern instance i_2 apparently includes i_1. The layout engine, therefore, automatically combines pattern instances where possible. However, the combination of pattern instances is pattern-dependent. For a pattern p that realizes a transitive relation on components (e.g., the alignment pattern p_A), two alignment pattern instances $\mathscr{I}(p, C_1)$ and $\mathscr{I}(p, C_2)$ are replaced by the pattern instance $\mathscr{I}(p_A, C_1 \cup C_2)$ if $C_1 \cap C_2 \neq \emptyset$.

```
proc computeAdHocLayoutWithPropagation(P, C_s, C_all)
  C = C_s
  do
    compute neighborhood C_n of C
    changed := computeAdHocLayout(p, C, C_n)
    C_c := components changed by the layout engine
    C := C ∪ C_c
  while changed
end
```

Listing 2: Algorithm for computing LAL/P

6 Future Work & Discussion

The integration of automatic ad-hoc layout into a diagram editor turns out to be a rather difficult task as performance issues as well as usability issues arise. Some of these issues are highlighted in the following, and some solutions are discussed.

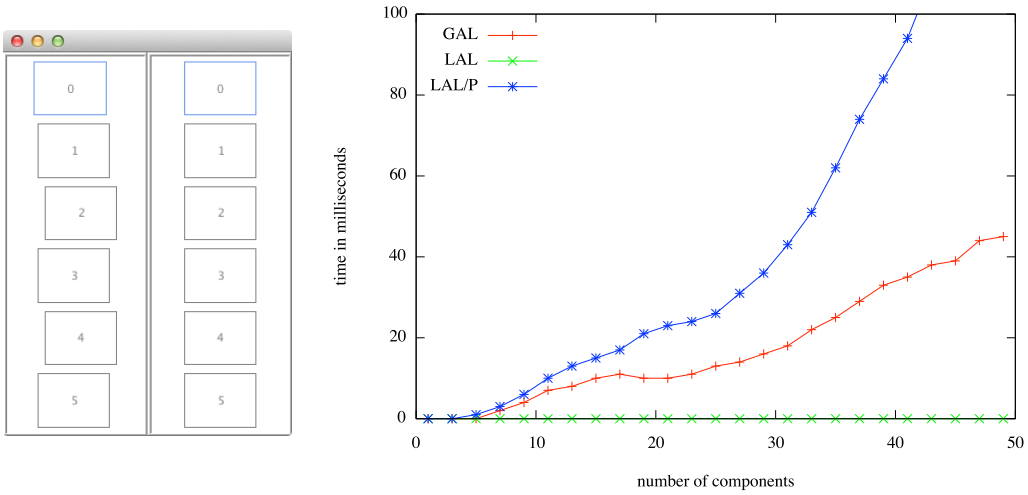
6.1 Performance

GAL rapidly results in performance issues, whereas LAL as well as LAL/P turned out to be practically usable. This is due to the fact that layout improvements are only computed locally, and hence performance does not (directly) depend on the size of the diagram, but rather depends on the structure of it.

In the following, two performance experiments are presented. Performance was measured on a machine equipped with an Intel Core i7 3.4 GHz processor, 8 GB RAM, running Mac OS X Version 10.7.2 and Java JDK 1.6.0_29. For the sake of simplicity, ad-hoc layout is computed for the vertical alignment pattern only. In addition, no pattern instances are present in the diagram before ad-hoc layout is computed.

The first performance experiment starts with a diagram that consists of n components that are arranged almost vertically. The user moves the topmost component (component 0) to the left. As a consequence, GAL as well as LAL/P would align all components vertically, whereas LAL would align the moved component and its lower neighbor only.

Figure 4 (a) shows an example: The diagram consists of $n = 6$ components, which are arranged almost vertically. The left figure shows the diagram after the user has moved component 0. The right figure shows the diagram after LAL/P was computed and the diagram was updated: All components are aligned vertically.



(a) Diagram (b) Performance

Figure 4: Automatic Ad-hoc Layout

Figure 4 (b) shows the time in milliseconds it takes to compute the GAL, the LAL and the LAL/P for a number n of components. As can be seen, the computation of LAL is very fast. The computation of GAL and LAL/P is more time-consuming. For instance, the computation of the LAL for $n = 40$ components takes about 1 millisecond, the computation of the GAL takes about 30 milliseconds, and the computation of the LAL/P takes about 90 milliseconds.

The second performance experiment starts with a diagram that consists of n components that are arranged as a matrix. The matrix is almost quadratic, e.g., a diagram with 100 components has 10 rows and 10 columns. The user moves the component in the top-left corner (component 0) to the left. As a consequence, GAL as well as LAL/P would align the leftmost column vertically, whereas LAL would align the moved component and its lower neighbor only.

Figure 5 shows an example: The diagram consists of $n = 40$ components, which are arranged as a matrix. The left figure shows the diagram after the user has moved component 0. The right figure shows the diagram after LAL/P was computed: The components of the leftmost column are aligned vertically now. The other components have not been modified.

Figure 6 shows the time in milliseconds it takes to compute the GAL, the LAL and the LAL/P for a number n of components. As can be seen, the computation of LAL is very fast. The computation of LAL/P takes a bit more time, but is still acceptable for the use in an interactive environment. In contrast, GAL is rather time-consuming. For instance, the computation of the LAL for the example shown in Figure 5 takes about 1 millisecond, the computation of the LAL/P takes about 5 milliseconds, and the computation of the GAL takes about 30 milliseconds.

The most striking difference between the results of the two performance experiments is that GAL performs better than LAL/P in the first experiment, while it is much slower in the second experiment. GAL performs better than LAL/P in the first experiment, because all components are involved in layout computation in both cases, and GAL does not need to take the neighborhood-relation of components into account. In the second performance test, LAL/P performs better than GAL, because only a small subset of components, namely the two leftmost columns, are involved in layout computation. In practical scenarios, usually only a small subset of components is involved in layout computation. Therefore, the first experiment can be considered as worst case scenario, while the second one as average case scenario.

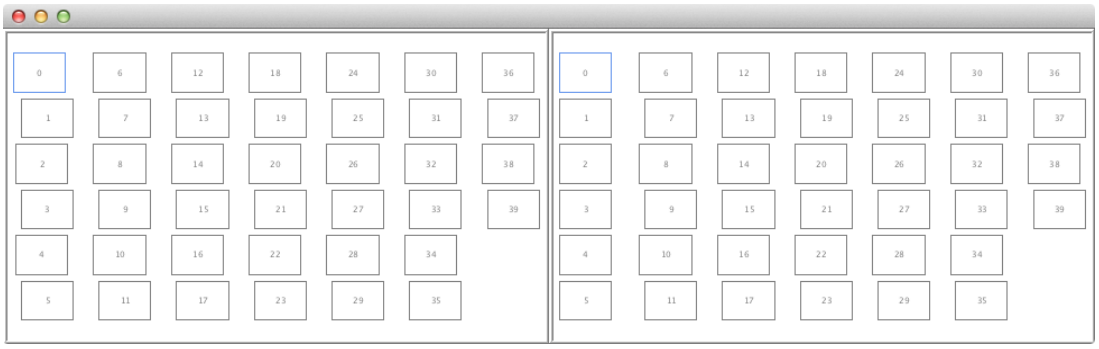


Figure 5: Diagram before and after Ad-hoc Layout computation

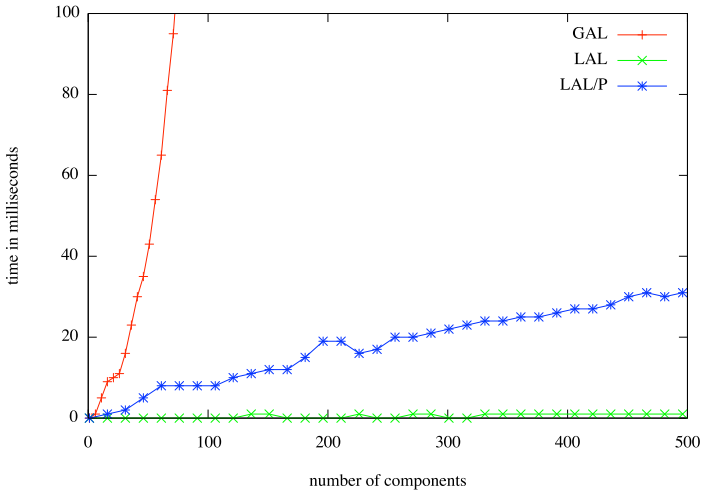


Figure 6: Performance of the example shown in Figure 5

6.2 Usability

GAL is quite powerful, but produces unpredictable layout improvements. LAL improves on this aspect, but only enables quite small layout improvements. LAL/P turned out to be a good compromise between predictability and power.

6.2.1 Pattern Instance Visualization

User studies have shown that the visualization of layout pattern instances is crucial in terms of usability enhancement. In Section 4, we described pattern instance visualization in the diagram and in the list at the bottom right of the editor. In the following, we describe how the visualization of pattern instances can be further enhanced.

At the moment, all pattern instances are visualized in the diagram as well as in the list at the bottom right of the editor. The current visualization does not distinguish pattern instances that were created by the user himself and pattern instances that were created in the course of automatic ad-hoc layout. A more sophisticated visualization could allow the user to distinguish these different “types” of pattern instances.

6.2.2 Additional Functionality

The new modes of operation, namely layout suggestions and automatic ad-hoc layout, make new functionality possible. Some of these features are discussed in the following.

Currently, ad-hoc layout is enabled for all patterns at once. To improve flexibility, one could also allow the user to choose the layout pattern(s) that are enabled.

Instead of applying layout patterns and updating the diagram, the layout improvement could be visualized in the diagram, first. Afterwards, the user could decide whether or not he wants to carry out these modifications. For the visualization, two variants are imaginable: Firstly, the

updated diagram could be shown in light gray on top of the diagram. Secondly, the new pattern instances could be visualized in the diagram, as it is done for user-controlled layout patterns, without updating the diagram itself. The second variant is similar to the way it is done in tools like POWERPOINT or VISIO.

At the moment, components that are in a "small neighborhood" of the selected components are chosen as input for the computation of the LAL and LAL/P. As an alternative, the following components could be chosen: components that are (completely) visible from one of the selected components, or components that are near a certain area, such as a vertical line, in case of the vertical alignment pattern.

With the approach at hand, it would be straightforward to define "snap to grid" and "hyper-snapping" as follows: After a user has modified a component, the layout engine updates this component in a sense that it is moved to the "correct" position, e.g. the top left edge of the component is "snapped" to the grid. This functionality contradicts the rule: "A layout engine should never modify a diagram component the user is currently modifying." As we want to stick to this rule, we did not include such functionality.

Alternatively, the layout engine could prevent the user from moving a component to a position that is not "correct", e.g., the left edge of the component does not lie on the grid. As this kind of functionality would restrict user interaction, our approach does not support such functionality.

7 Conclusions

In this paper, we have sketched our pattern-based layout approach, and the integration of the layout engine into diagram editors. We have focused on new modes of operation that are enabled by the pattern-based layout approach, namely layout suggestions and automatic ad-hoc layout.

So far, we have specified several layout patterns, and integrated them into various visual language editors. E.g., they have been integrated into the simple graph editor presented earlier, into a class diagram editor, and into a GUI forms editor that allows the user to create GUIs.

In addition, layout suggestions as well as automatic ad-hoc layout have been completely integrated into DIAMETA. Due to our pattern-based approach, the new modes of operation were made possible without extending the editor specification. However, the enhancements discussed in Section 6 have not yet been completely realized.

In practical tests, we have observed that the layout engine produces good results, and that the overall performance is satisfactory. They also showed that the implementation of features such as layout suggestions and automatic ad-hoc layout is feasible.

In the course of the integration of the new features, many questions arose concerning the usability of the editor. In the future, we will try to answer some of these questions, allowing us to choose the "right" functionality and to add the "right" enhancements. It is crucial to find the right balance between automatic layout, user-controlled layout, and automatic ad-hoc layout.

Bibliography

[BETT99] G. D. Battista, P. Eades, R. Tamassia, I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[BS86] E. A. Bier, M. C. Stone. Snap-dragging. In *Proc. 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'86)*. Pp. 233–240. ACM, 1986.

[DMS+08] T. Dwyer, K. Marriott, F. Schreiber, P. Stuckey, M. Woodward, M. Wybrow. Exploration of Networks using overview+detail with Constraint-based cooperative layout. *IEEE Transactions on Visualization and Computer Graphics* 14(6):1293–1300, 2008.

[DMW09] T. Dwyer, K. Marriott, M. Wybrow. Dunnart: A constraint-based network diagram authoring tool. In Tollis and Patrignani (eds.), *Proc. 16th Int. Symposium on Graph Drawing (GD'08)*. LNCS 5417, pp. 420–431. Springer, 2009.

[FKLD11] M. Frisch, S. Kleinau, R. Langner, R. Dachselt. Grids & guides: multi-touch layout and alignment tools. In *Proc. 2011 annual conference on Human factors in computing systems (CHI'11)*. Pp. 1615–1618. ACM, 2011.

[Mas01] T. Masui. HyperSnapping. In *Proc. IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*. Pp. 188–194. IEEE Computer Society, 2001.

[Min06] M. Minas. Generating Meta-Model-Based Freehand Editors. In Zündorf and Varró (eds.), *Proc. Int. Workshop on Graph-based Tools (GraBaTs'06), ECEASST*. Volume 1. 2006.

[MM07] S. Maier, M. Minas. A Pattern-Based Layout Algorithm for Diagram Editors. In Fish et al. (eds.), *Proc. Workshop on the Layout of (Software) Engineering Diagrams (LED'07), ECEASST*. Volume 7. 2007.

[MM10a] S. Maier, M. Minas. Combination of Different Layout Approaches. In Bottoni et al. (eds.), *Proc. 2nd Int. Workshop on Visual Formalisms for Patterns, ECEASST*. Volume 31. 2010.

[MM10b] S. Maier, M. Minas. Pattern-Based Layout Specifications for Visual Language Editors. In Bottoni et al. (eds.), *Proc. 1st Int. Workshop on Visual Formalisms for Patterns, ECEASST*. Volume 25. 2010.

[MM12] S. Maier, M. Minas. Integration of a Pattern-based Layout Engine into Diagram Editors. In Schuerr et al. (eds.), *Proc. 4th Int. Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE'11)*. LNCS 7233. Springer, 2012.

[RMS97] K. Ryall, J. Marks, S. Shieber. An interactive constraint-based system for drawing graphs. In *Proc. 10th annual ACM symposium on User interface software and technology (UIST '97)*. Pp. 97–104. ACM, 1997.

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.